COMS20001 lab. worksheet #1

- Both the hardware and software in MVB-2.11 is managed by the IT Services Zone E team. If you encounter a problem (e.g., a workstation that fails to boot, an error when you try to use some software, or you just cannot log into your account), they can help: either talk to them directly in room MVB-3.41, or submit a service request online via

  http://servicedesk.bristol.ac.uk
- We intend this worksheet to be attempted, at least partially, in the associated lab. session. Your attendance is important, since this session represents the primary source of formative feedback and help for COMS20001. Perhaps more so than in units from earlier years, *you* need to actively ask questions of and seek help from either the lectures and/or lab. demonstrators present.
- The questions are roughly classified as either L (for coursework related questions that should be completed in the lab. session), or A (for additional questions that are entirely optional). Keep in mind that we only *expect* you to complete the first class of questions: the additional content has been provided *purely* for your benefit and/or interest, so there is no problem with nor penalty for totally ignoring it (since it is not directly assessed).

Before you start work, download (and, if need be, unarchive[a]) the file

http://tinyurl.com/y72dnlcy/csdsp/os/sheet/lab-1_q.tar.gz

somewhere secure[b] in your file system: it is intended to act as a starting point for your own work, and will be referred to in what follows.

[a]Use the `gz` and `tar` commands within a BASH shell (e.g., in a terminal window), *or* the archive manager GUI (available either via the menu `Applications`→`Accessories`→`Archive Manager` or by directly executing `file-roller`) if you prefer.

[b]For example, the `Private` sub-directory in your home directory.

Q1[L]. This question introduces an (emulated) target platform which acts as a concrete instance of the example system covered in the lecture(s). It supplements the largely theory-based lecture(s) using a more practical, step-by-step overview of how software can be developed, executed, and debugged on the platform: this is important, since doing so is central to the coursework assignment.

Although the worksheet concludes with a set of challenges relating to experimental exploration, the more fundamental goal is to gain a solid understanding of the material. Put another way, the work you do will be biased toward reading and understanding rather than programming at this point. It is crucial *not* to view this as optional effort: carefully working through what is, admittedly, a detailed[1]worksheet will allow you to more easily and rapidly engage with later challenges.

The PB-A8 target platform

As in the lecture(s), we focus on a specific target platform represented by the RealView Platform Baseboard for Cortex-A8 [7] (which we refer to as PB-A8 for short). The PB-A8 is an *entire* computer system, so the complexity hinted at in [7, Figure 3-1], for example, should come as no surprise. However, do not let this put you off: in the main, we are interested only in

- the ARMv7-A [1], Cortex-A8 [4] processor core (labelled the "test chip" in [7]),
- the Universal Asynchronous Receiver/Transmitter (UART) whose ARM part number is PL011 [5],
- the timers whose ARM part number is SP804 [2],

plus a specific implementation of the ARM Generic Interrupt Controller (GIC) [3] design.

In the first half of the unit, you were challenged to develop software on an XMOS-based target platform. Contrast this with Figure 1, which illustrates a physical PB-A8: it should seem (and in fact is) more cumbersome to use due to the size and extra equipment needed to power and develop software for it. In an effort to a) reduce the challenge of software development, and b) address the issue of scale when used within the context of the unit (i.e., where there are > 100 students), we shift approach a little in this, second half: rather than a

[1]In an attempt to manage the level of detail, extended explanation of some topics is deferred to Appendix A and Appendix B. Having an Appendix in a worksheet might seem, and possibly is ridiculous, but the goal is to offer help to those who need it *without* also cluttering the worksheet itself too much. Hopefully this seems a reasonable compromise.

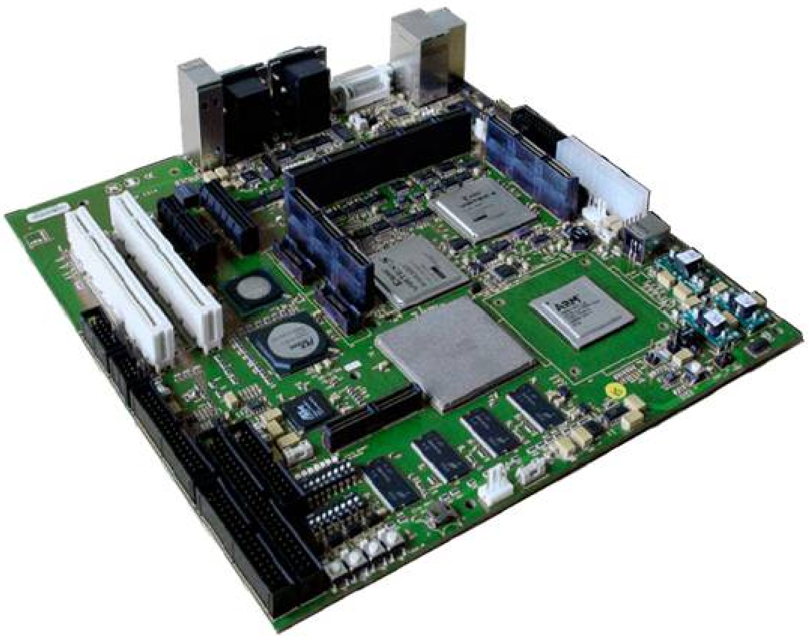

Figure 1: *A PB-A8 target platform.*

physical PB-A8, we use QEMU[2] to *emulate* the same target platform with development carried out using (and the emulator executed on) a standard Linux-based lab. workstation.

The "hello world" kernel image

In order to introduce the PB-A8 platform, the following Sections act as a guide to controlled execution of an example, "hello world" kernel image. Although this is not a *kernel* per se, we retain the term because a) QEMU uses the same, generic term for any image (or program) provided for it to execute, but, perhaps most importantly, b) it stresses the fact that execution will be on bare-metal, and thus in a privileged processor mode (akin to a kernel).

Explore the archive content

The first step is to explore the example. Assuming it is unarchived into some path referred to from here on as `${ARCHIVE}`, Figure 2 shows the structure of the material provided. It can be viewed as having two parts:

a Some source code organised in three directories, namely

- `device`, which supports[3] access to pertinent devices on the PB-A8, using various definitions to model the memory map,
- `kernel`, which contains kernel mode related source code, and
- `user` which contains user mode related source code,

plus the linker script `image.ld`, which are combined to produce a kernel image QEMU can execute.

b A build system for the source code, namely a `Makefile` with six targets:

- `build` uses `gcc` et al. to compile and link all source code files into a kernel image ready for use,
- `launch-qemu` launches an instance of `qemu-system-arm`, which loads the (pre-)compiled kernel image and waits for a remote debugger to connect,
- `launch-gdb` launches an instance of `gdb`, connecting to the waiting instance of `qemu-system-arm` st. it then controls execution,
- `kill-qemu` terminates any/all instances of `qemu-system-arm`,

[2]See http://www.qemu.org, http://en.wikibooks.org/wiki/QEMU, and/or [8].

[3]This source code is provided as a layer of abstraction, in an attempt to minimises the amount of low-level detail you are exposed to. You *could* attempt to understand and even extend it to suit your requirements, but equally there is no problem with simply using it as a form of "platform API" and ignoring how it works. Since the relevant content differs between worksheets, Figure 2 refers to it as `*.h` and `*.c`, i.e., "a set of *some* header and source code files".

- `kill-gdb` terminates any/all instances of `gdb`, and
- `clean` removes any compiled material (e.g., any intermediate object files, plus the kernel image).

Understand the archive content

In an attempt to understand what the source code itself *does*, the natural next step is to examine the archive content in some detail. In this example there are only a few files of interest, each of which is explained line-by-line in what follows.

`image.ld`: the linker script Figure 3 illustrates the linker script `image.ld`. It controls how `ld` produces the kernel image from object files, which, in turn, stem from compilation of the source code files; the resulting layout in memory is illustrated by Figure 4.

- In a linker script, you can interpret the special "full stop" (or "dot") symbol as the current address during the layout of content in memory. As such, Line #10 assigns the current, and so initial address to $70010000_{(16)}$. Effectively, this means content from the kernel image is laid out assuming it will be loaded into memory by QEMU starting at address $70010000_{(16)}$.
- Lines #12, #14 and #16 place the text, data and bss segments of *all* object files in a subsequent region of addresses. Notice that we explicitly place the text segment from `lolevel.o` before all the others, and, given it is therefore the first content to be placed overall, it will be loaded into memory starting at address $70010000_{(16)}$.
- Line #18 aligns the current address to an 8-byte boundary, matching what the AAPCS [6] function calling convention expects.
- Lines #20 and #21 assign the symbol `tos_svc` a value equal to the current address, having incremented it by 4KiB: this is intended to allocate a 4KiB region for use as a stack when the processor is in SVC mode. Note `tos_svc` initially points at the *top* of, or *largest* address in said stack: the stack will grow *downward* from this point (so toward the other content in memory, e.g., the bss segment).

`lolevel.[sh]`: low-level kernel functionality All low-level, kernel-specific functionality is captured by the header file `lolevel.h` and source code `lolevel.s`. The former is uninteresting, in the sense it is empty. The idea is that it could (in the future) be used to declare prototypes matching assembly language functions implemented in `lolevel.s`; by including the header file, `hilevel.c` could then "see" and therefore invoke said functions. The latter implements the function `lolevel_handler_rst`:

- Line #10 moves the literal $D3_{(16)}$ into the CPSR register, implying that $\mathsf{CPSR}[M] = 10011_{(2)}$, $\mathsf{CPSR}[F] = 1_{(2)}$, and $\mathsf{CPSR}[I] = 1_{(2)}$. That is, the processor is switched into SVC mode with both FIQ and IRQ interrupts disabled. Remember this means any subsequent reference to `sp`, for example, will resolve to the banked, SVC mode version of that register.
- Line #11 initialises the SVC mode stack, setting the SVC mode stack pointer to `tos_svc` (as specified in the linker script).
- Line #13 invokes the high-level, C function `hilevel_handler_rst` defined in `hilevel.c`.
- Line #14 realises an infinite loop: the `b` instruction branches to *itself*. This might seem odd, but keep in mind that there is nowhere sane[4] for `lolevel_handler_rst` to return to once complete, so the loop is intended to model halting the processor.

When QEMU is executed, it in fact *always*[5] loads the kernel image into memory at address $70010000_{(16)}$ and transfers control to this entry point using a bootloader[6] placed at address $7000000_{(16)}$ which is where execution *actually* begins. When using the example kernel image, `lolevel_handler_rst` is therefore invoked when QEMU is executed: *it* was placed at address $70010000_{(16)}$ by the linker, because a) we placed the content of `lolevel.o` first, and b) `lolevel_handler_rst` is the first function defined in it. In turn, it invokes `hilevel_handler_rst` once the stack pointer for SVC mode is initialised.

[4]The reason for this is simple: `lolevel_handler_rst` is only invoked as a consequence of the processor being reset (e.g., at power-on), rather than by some other function.

[5] This stem from launching QEMU using the `-kernel` option; a a longer, more concrete explanation is provided by reading through http://www.github.com/qemu/qemu/blob/master/hw/arm/realview.c, working backwards from Line #366.

[6]http://wiki.osdev.org/Bootloader

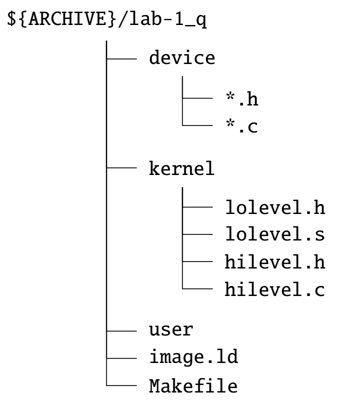

Figure 2: *A diagrammatic description of the material in* `lab-1_q.tar.gz`*.*

```
SECTIONS {
  /* assign load address (per  QEMU) */
  .        =      0x70010000;
  /* place text segment(s)           */
  .text : { kernel/lolevel.o(.text) *(.text .rodata) }
  /* place data segment(s)           */
  .data : {                         *(.data        ) }
  /* place bss  segment(s)           */
  .bss  : {                         *(.bss         ) }
  /* align       address (per AAPCS) */
  .        = ALIGN( 8 );
  /* allocate stack for svc mode     */
  .        = . + 0x00001000;
  tos_svc = .;
}
```

Figure 3: *The linker script* `image.ld`*.*

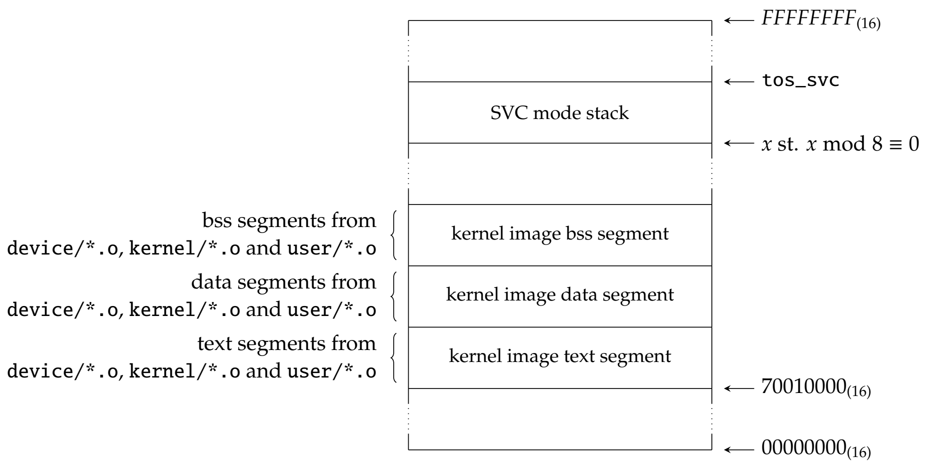
FFFFFFFF(16)
tos_svc
SVC mode stack
x st. x mod 8 ≡ 0
bss segments from device/*.o, kernel/*.o and user/*.o
kernel image bss segment
data segments from device/*.o, kernel/*.o and user/*.o
kernel image data segment
text segments from device/*.o, kernel/*.o and user/*.o
kernel image text segment
70010000(16)
00000000(16)

Figure 4: *A diagrammatic description of the memory layout realised by* `image.ld`*.*

`hilevel.[ch]`: high-level kernel functionality All high-level, kernel-specific functionality is captured by the header file `hilevel.h` and source code `hilevel.c`. The former is uninteresting bar the fact it includes headers that allow access to memory-mapped I/O device on the PB-A8. The latter implements a high-level interrupt handler function `hilevel_handler_rst`:

- Lines #14 to #16 realise an inner `for` loop which iterates through each `i`-th character in the string `x`: each one is transmitted via the `PL011_t` instance `UART0` by invoking `PL011_putc`. Note that the Boolean flag passed to `PL011_putc` indicates it should block, i.e., wait until transmission is possible and hence completed before returning.
- Lines #13 to #17 realise an infinite outer `while` loop, each iteration of which thus transmits `x`, i.e., "hello world", via `UART0` per the above. This implies control-flow will never reach Line #19, so the function will never return.

The `launch-qemu` target in `Makefile` associates an emulated UART[7] with the standard streams (i.e., `stdin` and `stdout`) of the QEMU process. Put simply, this allows software executing on the emulated target platform to interact with the development platform. For example, transmitting a character via the `PL011_t` instance `UART0` results in writing it via `stdout` and thus the emulation terminal.

Experiment with the archive content

Now, finally, you are ready to actually do something! Each step in what follows relates to executing the example kernel image under control of `gdb`, and, as a result, some valuable hands-on experience. Start by launching three terminals which will be referred to as the

- development terminal (this is where you edit source code, e.g., using `emacs`),
- debugging terminal (this is where you execute `gdb`), and
- emulation terminal (this is where you execute QEMU, i.e., `qemu-system-arm`)

respectively.

Challenge #1: build the image Build the kernel image by issuing the command

```
make build
```

in the development terminal: you should find that `image.bin` and `image.elf` have been produced as a result.

Challenge #2: execute the image

a First launch[8] QEMU, by issuing the command

```
make launch-qemu
```

in the emulation terminal. QEMU has been instructed to wait for a connection from a debugger, so next issue the command

```
make launch-gdb
```

in the debugging terminal. A `gdb` prompt replaces the shell prompt, indicating that `gdb` is ready to accept commands and hence control QEMU.

b At this point you are ready to have QEMU execute the kernel image, so can issue the command

```
continue
```

in the debugging terminal: `gdb` resumes execution from whatever address the program counter holds (e.g., where execution was started initially, or was last halted). You *should* see "hello world" written (continuously) to the emulation terminal, matching what we expect from having examined the source code.

[7] http://en.wikipedia.org/wiki/Universal_asynchronous_receiver/transmitter

[8] At least for the QEMU installation on the lab. workstations, you *may* see warnings similar to `pulseaudio: set_sink_input_volume() failed` and/or `pulseaudio: set_sink_input_mute() failed`. You can ignore them, and proceed as normal: they are QEMU noting it failed to initialise pulseaudio (which, as the name suggests, is a system for managing audio devices and so playback of sound) for some reason, but you will not use this functionality.

c Once you are fed up, the final step is to instruct gdb to halt execution. However, there is currently no gdb prompt shown: QEMU has not yet returned control to gdb after you last instructed it to continue execution. So you need to forcibly interrupt execution by pressing (and holding) Ctrl-C in the debugging terminal until a gdb prompt is again shown. Now issue the command

```
quit
```

in the debugging terminal so gdb terminates, and you get a shell prompt back. Finally, force termination of the QEMU instance by issuing the command

```
make kill-qemu
```

again in the debugging terminal *or* pressing (and holding) Ctrl-C in the emulation terminal.

Challenge #3: execute the image under more control

a Launch QEMU and gdb by repeating associated steps from above.

b Rather than continue indefinitely, as above, gdb also enables you to continue execution until a specific breakpoint (an instruction or statement) is reached *or* a condition is met. To see this in action, issue the commands

```
break hilevel_handler_rst
```

then

```
continue
```

in the debugging terminal. The first instructs gdb to halt execution once it reaches hilevel_handler_rst, and the second resumes execution as before. However, rather than again having to interrupt execution to get the gdb prompt back, this time gdb offers the prompt: the breakpoint was reached, so execution halts. Note that you can have more than one breakpoint active at any one time, and that

```
info breakpoints
```

will list them for you; deleting a breakpoint from the list can be accomplished with

```
delete breakpoints 1
```

where the numeric argument identifies said breakpoint (in this case the breakpoint numbered 1, corresponding to that created above).

c gdb enables you to single-step execution, meaning it it offers the prompt after executing either the next instruction or statement: this is accomplished via the stepi and step commands respectively. Try this out: execution was previously halted at hilevel_handler_rst, so now issue the command

```
step
```

in the debugging terminal to single-step one statement further. You can repeat the previous command simply by issuing an empty command, i.e., by pressing return, so do this say 10 times. You should see execution progress through PL011_putc, writing the first few characters of "hello world". Note that the command

```
step 10
```

will single-step though the next 10 statements in one go if you prefer, with an analogous syntax for stepi.

d Terminate QEMU and gdb by repeating associated steps from above.

Challenge #4: execute the image displaying more information

a Launch QEMU and gdb by repeating associated steps from above.

b Using a breakpoint as above, control execution st. it halts once the hilevel_handler_rst function is invoked.

c gdb supports various mechanisms for what could be generically described as inspecting the state of execution.

- The `x` (or "examine") command is used to show the content of memory starting at a given address: various formats can be used, e.g., interpreting said content as a sequence of 8-bit decimal bytes, or 32-bit hexadecimal words. Try this out: issue the command

  ```
  x/16i 0x70010000
  ```

  and then

  ```
  x/16x 0x70010000
  ```

  in the debugging terminal. The two commands show memory content at address $70010000_{(16)}$, yielding 16 (disassembled) instructions in the first instance and then 16 hexadecimal words (i.e., the encoded instructions) in the second instance.
- Rather than inspect memory via an address, it can be more convenient to display the value of a variable (as defined in some C function, for example: there is no analogy in assembly language). Issue the command

  ```
  print x
  ```

  in the debugging terminal: this shows both the address of the variable `x`, *and* the value. Note that the alternative

  ```
  display x
  ```

  does more or less the same, except that it is "sticky" in the sense the output is repeated after every command.
- The `disassemble` command is what it sounds like: it disassembles the machine code in memory, and shows what the assembly language equivalent would look like. Try this out: issue the command

  ```
  disassemble hilevel_handler_rst
  ```

  in the debugging terminal: you should be able to identify both the outer (infinite) `while` loop and inner `for` loop, plus the call to `PL011_putc` in the ~ 11 instruction assembly language disassembly of the `hilevel_handler_rst` function.
- Since execution is halted, `gdb` can also inspect the state of the processor. For example, one might issue the command

  ```
  info registers
  ```

  to show the content of the ARM register file, or

  ```
  info stack
  ```

  to describe the entire stack, or

  ```
  info frame
  ```

  to describe the current frame on that stack.

d Terminate QEMU and `gdb` by repeating associated steps from above.

Next steps

There are various things you could (optionally) do next. The most obvious is to exercise the capability you now have to execute ARM assembly language programs via QEMU: in essence, you could replace `lolevel_handler_rst` with *anything*. As such, this represents an opportunity to experiment with your *own* ARM assembly language programs. If you need a specific challenge, attempt to write a solution for a previous coursework. Or, try to solve something from

http://projecteuler.net/

using assembly language.

If you need some help taking this next step, it can make sense to consider various tools other than QEMU. In particular, consider making use of

a a tool at

http://gcc.godbolt.org

which provides a web-interface to an ARM-based GCC compiler, thereby allowing easy exploration of how high-level C programs are compiled into low-level assembly language programs, and b a tool at

http://salmanarif.bitbucket.org/visual

which offers various user-friendly ways to explore the execution of (emulated) ARM assembly language programs

as a way to explore and so improve your understanding of related concepts.

Q2[A]. In a nutshell, system programming[9] is the task of producing system software: unlike application software, it typically a) operates at a lower level of abstraction, often interacting more directly with the kernel and/or hardware devices say, b) is performance critical, and c) provides functionality to other software rather than (human) users. Various forms of documentation can be helpful, with UNIX-style manual pages[10] often representing a canonical source of information for system calls etc. However, these are often written in a fairly formal and technical style, so, although acting as an effective reference, they are normally less ideal as a learning resource. Fortunately, a range of accessible alternatives are available online. The set of guides at

http://beej.us/guide

represents a good example.

a You may feel a need to recap on your knowledge of C programming. In addition to material encountered during your study of this topic, the guide at

http://beej.us/guide/bgc

might be useful in offering an alternative perspective.

b `gdb` is an extremely powerful tool; the introduction above covered a (very) limited sub-set of the functionality that it provides. As such, time *invested* in learning about `gdb` will pay off later wrt. the time *saved* when debugging your programs. In addition to the guide at

http://beej.us/guide/bggdb

some other resources you could use include the online documentation at

http://www.gnu.org/software/gdb

or various online tutorials such as

http://www.youtube.com/watch?v=sCtY--xRUyI

or even dedicated books [10, 9].

References

[1] ARM Limited. *ARM Architecture Reference Manual: ARMv7-A and ARMv7-R edition*. Tech. rep. DDI-0406C. http://infocenter.arm.com/help/topic/com.arm.doc.ddi0406c/index.html. 2014 (see p. 1).

[2] ARM Limited. *ARM Dual-Timer Module (SP804) Technical Reference Manual*. Tech. rep. DDI-0271D. http://infocenter.arm.com/help/topic/com.arm.doc.ddi0271d/index.html. 2004 (see p. 1).

[3] ARM Limited. *ARM Generic Interrupt Controller Architecture Specification (GIC architecture version* 3.0 *and version* 4.0*)*. Tech. rep. IHI-0048B. http://infocenter.arm.com/help/topic/com.arm.doc.ihi0048b/index.html. 2015 (see p. 1).

[4] ARM Limited. *Cortex-A8 Technical Reference Manual*. Tech. rep. DDI-0344K. http://infocenter.arm.com/help/topic/com.arm.doc.ddi0344k/index.html. 2010 (see p. 1).

[5] ARM Limited. *PrimeCell UART (PL011) Technical Reference Manual*. Tech. rep. DDI-0183F. http://infocenter.arm.com/help/topic/com.arm.doc.ddi0183f/index.html. 2005 (see p. 1).

[6] ARM Limited. *Procedure Call Standard for the ARM Architecture*. Tech. rep. IHI-0042E. http://infocenter.arm.com/help/topic/com.arm.doc.ihi0042e/index.html. 2012 (see pp. 3, 12).

[9] http://en.wikipedia.org/wiki/System_programming

[10] The `man` (or manual) command can be used to display said documentation via the command line. Two options are extremely useful: the `-k` option supports search for a keyword, and the `-s` option supports specification of a section within the manual (and hence disambiguation of keywords occurring in multiple entries). For example, `man -k send` will search for all entries including the keyword `send`, and `man -s 2 send` will display the specific entry within section 2.

[7] ARM Limited. *RealView Platform Baseboard for Cortex-A8*. Tech. rep. HBI-0178. http://infocenter.arm.com/help/topic/com.arm.doc.dui0417d/index.html. 2011 (see p. 1).

[8] F. Bellard. "QEMU, a fast and portable dynamic translator". In: *USENIX Annual Technical Conference (ATEC)*. 2005, pp. 41–46 (see p. 2).

[9] N. Matloff and P.J. Salzmann. *Art of Debugging with GDB and DDD*. No Starch Press, 2008 (see p. 8).

[10] R.M. Stallman, R. Pesch, and Shebs S. *Debugging with GDB: The GNU Source-Level Debugger*. GNU Press, 2002 (see p. 8).

Models of software development

Some people find it hard to understand what emulation *is*, or, put another way, what relationship exists between the physical and emulated target platforms. In an *attempt* to explain, and although heavily dependant on the specific context and objectives, one can consider (at least) three scenarios when developing some software artefact:

- Figure 5 illustrates scenario #1, which you are likely most familiar with: the idea is that one first a) develops a program, say `foo.c`, which is b) compiled into an executable `a.out` then c) executed. Each step is performed on the same platform, which is to say the development and target platforms are the same. You might be so used to using this approach that various subtle yet important facts seem trivial. For example, note that

  – the tool-chain used for compilation is native, which means it generates machine code for the same platform it executes on,
  – whenever a program is executed, an associated process will be created and managed by (i.e., will execute under control of) an operating system kernel, and
  – such a process can interact with a rich set of resources: it can usually write to and read from a file system, for example, and use standard streams (e.g., `stdout`) as a way to communicate with both the user and other processes.

  Such an approach has no obvious disadvantages per se, in the sense it is, by design, a very convenient way to develop software.

- Figure 6 illustrates scenario #2, where target and development platforms are now physically separate. A common reason for this separation is they differ in type: if the target platform is an embedded device, for example, it may be too constrained (there is often less memory available, and the processor is less computationally able) or lack resources required to host a kernel. As a solution, it is common to develop software on the development platform and then transfer this for bare-metal execution on the separate target platform. A range of implications stem from this difference:

  – the tool-chain used will differ somewhat: the compiler used is now a cross-compiler (in the diagram, a compiler for X st. we use `x-gcc` not `gcc`) meaning it executes on the development platform but generates machine code for the target platform,
  – because there is no kernel, the programmer is tasked with how a) the program is loaded into memory on the target platform, b) execution is then involved, *and* c) any subsequent communication with the development platform, and
  – the executing program has full control over the platform, but no support (other than from any statically linked libraries): for example, there simply is no `stdout` to use in the same way as scenario #1.

  This approach provides some advantages: bare-metal execution allows use of instructions or resources that may otherwise be protected by the kernel, for example. However, it can also present significant development challenges, such as the necessity for use of a remote debugger.

- Figure 7 illustrates scenario #3, which represents a compromise between scenarios #1 and #2. In a sense scenario #3 is the same as scenario #2, because the target and development platforms differ. However, the physical target platform is replaced by an emulated (or simulated) alternative.

  The advantage of doing so is that the (software) emulator executes on, and can therefore interact with the development platform as with scenario #1. So although we still need a cross-compiler and so on to develop software, some of the development challenges presented by scenario #2 are lessened. The (potential) disadvantage is realism, in the sense that accuracy of the emulator is now crucial. If it were *in*accurate our software might behave differently on the physical and emulated target platforms, which is less than ideal. Provided it *is* accurate, however, the software we develop is no less valid than in scenario #2. Rather, it simply removes the need for a physical target platform until the software is then later deployed.

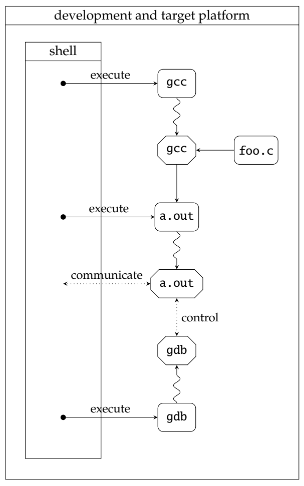

Figure 5: *A block diagram attempting to illustrate development scenario #1: since the development and target platforms are the same, this should reflect your normal compile-execute-debug cycle as invoked from a shell or via an IDE.*

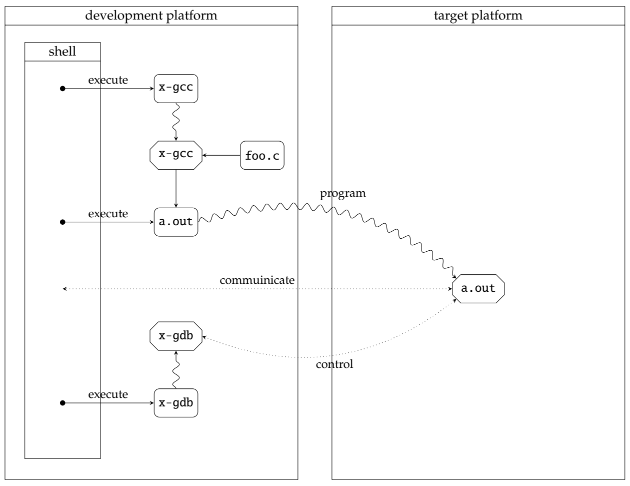

Figure 6: *A block diagram attempting to illustrate development scenario #2: since the development and target platforms differ, there needs to be a) an explicit programming step (to transfer* `a.out` *into memory on the target platform), and b) extra physical connectivity (e.g., via USB or similar) to allow communication and remote debug.*

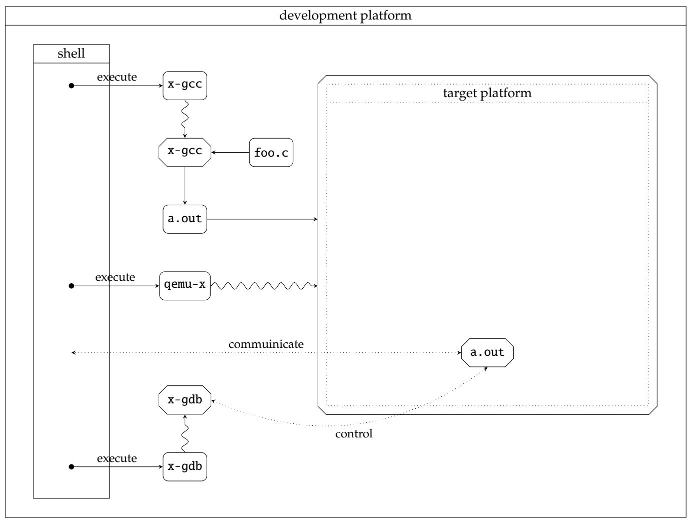

Figure 7: *A block diagram attempting to illustrate development scenario #3: the emulated target platform is now shown as a process executing on the development platform, thus acting as a form of intermediate solution that blends features of Figure 5 and Figure 6.*

Developing software for the PB-A8 using QEMU

Developing then executing and debugging software for the PB-A8 can be challenging, in the sense doing so will differ from your normal approach. As such, the subsequent Sections attempt to clearly explain a) what exactly each difference is, b) what options exist for coping with said differences, and c) which option is assumed here (and therefore recommended for use in the coursework assignment.

Writing programs The nature of software we are interested in developing is different from the norm. More specifically, it will often *require* the use assembly language, e.g., to guarantee control over what is executed, and/or access a low-level processor feature which lacks support at a higher level. Faced with this requirement, (at least) two strategies are possible: you could

- write a high-level part in C, and link it to any low-level part(s) written in *raw* assembly language, or
- write a high-level part in C, and embed any low-level part(s) in that program using *inline*[11] assembly language.

We will a) assume use of the first strategy, since it enforces a cleaner separation between the high- and low-level parts, but b) try to minimise the amount of assembly language *you* have to write yourself.

Compiling programs Either way, compiling the resulting program demands more care than usual. Whereas normally the default tool-chain options will suffice, we need to carefully set those options to ensure the compilation process matches our requirements. With this in mind, we assume `gcc` is invoked as follows:

- It is important to be clear about which processor will be targeted, since the compiler may behave (e.g., make an optimisation decision) differently for one vs. another. As such, we use

  ```
  -mcpu=cortex-a8
  ```

  to specify the exact processor model.
- It is important to be clear about which function calling convention will be used, otherwise compiled and hand-written assembly language programs cannot interact correctly. This is achieved by
  - *forcing* use of the ARM-specified AAPCS [6] by using `-mabi=aapcs`, and
  - assuming the `-fomit-frame-pointer` is used either explicitly or implicitly (i.e., as enabled by the `-O` optimisation flag); the frame pointer register (which is optional under AAPCS) is not used as a result, so the calling convention is simpler.

Linking programs

1. As already outlined by Appendix A, a bare-metal, unhosted target platform differs from a hosted alternative wrt. the support it provides during execution of software. Within this context, the C standard library[12] represents an central component. In general, it could be thought of as performing two roles: it

   (a) provides a range of kernel-agnostic functionality relating to the C language (e.g., `stdint.h`, which just defines types such as `uint32_t`), and

   (b) provides a range of kernel-specific functionality, and thus a software layer to abstract interaction with the kernel (e.g., `stdio.h`, which defines I/O functionality and thus depends on the implementation of I/O in the kernel).

   The latter is a problem for an unhosted platform since there is no kernel to interact with, but clearly *some* functionality will be useful even in this case. Therefore, various limited (or pared down) implementations of the library exist: an example is

   http://www.sourceware.org/newlib

   which we assume use of here.

[11] http://wiki.osdev.org/Inline_Assembly
[12] http://wiki.osdev.org/C_Library

2. The kernel is normally tasked with management of each process which results from execution of an associated program. An important part of this task, as performed by the loader, is initialisation of the address space: an obvious example is the text segment, populated by instructions from the executable image that constitute the program, which may or may not be subjected to some form of relocation.

   On an unhosted target platform, there is no kernel so *we* must take responsibility for this task. Doing so involves provision of extra information to the linker `ld`, in the shape of an explicit (rather than implicit, default) linker script[13]. We assume `ld` is invoked with the

   ```
   -T image.ld
   ```

   option where `image.ld` is the linker script in question.

Debugging programs Debugging software can be difficult at the best of times; whenever you are faced with doing so, a debugger can be invaluable. For example, executing a program under control of `gdb` allows single-stepping through individual instructions or direct inspection of their influence on the registers and/or memory.

`gdb` supports local or remote debugging, capturing cases where the development and target platforms are the same or differ respectively. In the latter case, a `gdb`-based interface (or shell) executed on the development platform can "remote control" the target platform via a communication link (e.g., a physical cable, or across a network). We will assume this strategy is employed, and specifically that:

- Compilation and assembly using `gcc` and `as` uses the `-g` flag to produce output with debugging information (e.g., symbols) included; this allows `gdb` to form a correspondence between the source code and machine code.
- `qemu-system-arm` is executed using
  - the `-gdb tcp:127.0.0.1:1234` option so a `gdb`-friendly remote debugging interface is presented on TCP port 1234 of the development platform, and
  - the `-S` flag to halt execution once the emulated platform is initialised (i.e., it waits st. `gdb` can connect before any instructions are actually executed)

  meaning `gdb` can subsequently connect to and remotely debug the program *it* is executing.

[13]http://wiki.osdev.org/Linker_Scripts